Control-Flow Integrity

University of Texas at Dallas
Language Based Security
CS 6v81 - 002

January 29, 2008

Problem

- Many current attacks abuse exploits to subvert machine-code execution
  - buffer overflow attacks
  - “jump-to-libc” attacks
  - “pointer subterfuge” attacks
- Vulnerability mitigations deployed thus far are circumventable by attackers
  - *stack canaries, runtime elimination of buffer overflows, randomization and artificial heterogeneity, and tainting of suspect data*

Solution

- Control-Flow Integrity (CFI):
  - Execution of a program dynamically follows only certain paths, in accordance with a static policy (a Control-Flow Graph)
  - Dynamic checks & machine code rewriting
- Control-Flow Graph (CFG):
  - defined by analysis ahead of time
    - source code analysis, **binary analysis**, execution profiling

Enforcement

- Valid destinations determined by CFG
- Constant destinations are trivial
- Dynamic destinations require a dynamic check
  - Machine-code rewriting
  - ID (bit pattern) inserted at each destination
  - Dynamic ID-check inserted before each source
    - Ensure runtime destination has proper ID equivalence class
  - Static inspection to verify machine-code rewriting

Example CFG

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

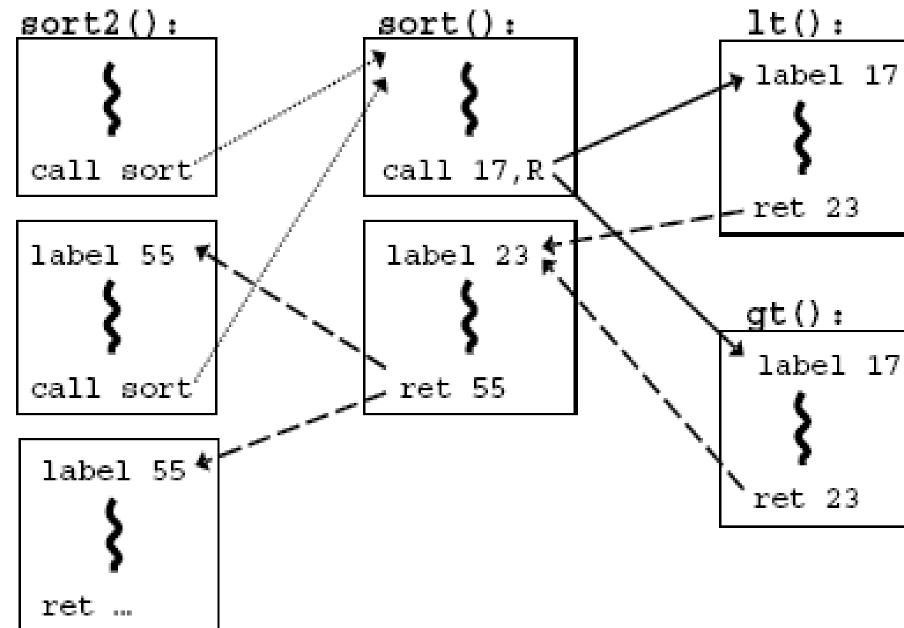

Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

- Destination Equivalence
  - Code duplication
  - Refining the instrumentation

So, what does CFI do?

- Ensures that runtime execution proceeds along a given CFG
- Guarantees execution of a function starts at the beginning and proceeds beginning to end
- Assumes the attacker is persistent and can arbitrarily change:
  - data memory
  - most registers
- Prevent the circumvention of IRMs and SFI

Then, what doesn't it do?

- Provide fault tolerance
- Ensure a function call returns to the most recent callsite invoking the function
- Prevent exploits within the bounds of the CFG
  - Exploits that rely on incorrect argument-string parsing to improperly launch an executable

Example CFI Instrumentation

Source				Destination		
Opcode bytes	Instructions			Opcode bytes	Instructions	
FF E1	jmp ecx		; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst
				...		

can be instrumented as (a):

81 39 78 56 34 12	cmp [ecx], 12345678h	; comp ID & dst	78 56 34 12	; data 12345678h	; ID
75 13	jne error_label	; if != fail	8B 44 24 04	mov eax, [esp+4]	; dst
8D 49 04	lea ecx, [ecx+4]	; skip ID at dst	...		
FF E1	jmp ecx	; jump to dst			

or, alternatively, instrumented as (b):

B8 77 56 34 12	mov eax, 12345677h	; load ID-1	3E 0F 18 05	prefetchnta	; label
40	inc eax	; add 1 for ID	78 56 34 12	[12345678h]	; ID
39 41 04	cmp [ecx+4], eax	; compare w/dst	8B 44 24 04	mov eax, [esp+4]	; dst
75 13	jne error_label	; if != fail	...		
FF E1	jmp ecx	; jump to label			

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

Assumptions

- Unique IDs (UNQ)
  - ID bit patterns must not be present anywhere in code other than IDs and ID-checks
- Non-Writable Code (NWC)
  - Program cannot modify code memory at runtime
    - ID-checks can be circumvented
- Non-Executable Data (NXD)
  - Program cannot execute data as if it were code
    - Attacker can execute data labeled with expected ID

Phases of Inlined CFI Enforcement

1) Construct CFG
2) CFI instrumentation
3) Establish UNQ assumption
4) CFI verification, validates the following:
   - Direct jumps and similar instructions
   - Proper insertion of IDs and ID-checks
   - UNQ property

Performance Results

- Overhead is competitive with or better than cost of most comparable techniques
- CFG construction + CFI Instrumentation took ~10 seconds
- Binary size increased by ~8%
- Benchmarks took ~16% longer to execute on average

Expanding CFI

- Faster SFI
- SMAC
- Protected Shadow Call Stack

Faster SFI

- CFI makes current optimizations more robust and enables new ones
- Guarantees about control flow remove the need to check memory addresses in local variables repeatedly
- Massive overhead reduction
  - Ex: 4.7% overhead rather than 23% for MD5

SMAC: Generalized SFI

- Extension of SFI that allows different access checks to be inserted at different instructions in the untrusted code
  - Ex: isolated data memory regions that are only accessible from a particular part of the code

SMAC & CFI

- SMAC can help eliminate CFI assumptions
  - Non-Writable Code (NWC)
    - Disallow writes to certain memory addresses
  - Non-Executable Data (NXD)
    - Prevent control flow outside those addresses

Protected Shadow Call Stack

- Increases precision of CFI enforcement
  - Ensures a function call returns to the most recent callsite invoking the function
- Assumptions:
  - Attacker cannot modify this stack directly
  - Stack is guarded against corruption resulting from program execution

Future Work

- Investigate attractive hardware CFI implementations
- Research methods of preventing exploits within the bounds of the CFG

References

- Martín Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *ACM Conference on Computer and Communication Security,* Alexandria, VA, November 2005.
- Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A Theory of Secure Control-Flow. *International Conference on Formal Engineering Methods,* Manchester, UK, November 2005.

Discussion Questions

- What is the trusted minimal computing base in CFI?
- Why isn't a buffer overflow attack possible?